PR #37026 完整报告

sgl-project/sglang

fix(hicache): isolate decode offload state per request

合并时间：2026-08-30 09:15

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/37026

执行摘要

- 一句话：decode offload 状态改按 Req 实例隔离，修复 rid 重用竞态
- 推荐动作：值得精读。这是一次教科书式的异步生命周期键选择修复：通过将簿记键从“外部可重用的身份标识”切换为“内部持有确切引用的对象实例”，从根本上消除了跨请求状态串扰，且改动面极小。建议关注三点：一是 Req 对象作为字典键的哈希语义约定；二是 _check_offload_progress 中 ack 处理顺序与 finish_event.synchronize() 的配合；三是回归测试“先构造同 rid 双请求、再让迟到 ack 到达”的写法，可作为类似竞态复现的模板。

功能与动机

PR body 明确指出：DecodeKVCacheOffloadManager tracks asynchronous D2H offload state by the caller-provided rid. A completed request is removed from rid_to_state before its copy necessarily finishes, so a new request reusing the same rid can inherit stale progress and be modified by a late callback from the old request. 作者还给出了脱离 GPU 时序的确定性复现：仅将回归测试恢复到 base 提交，test_reused_rid_does_not_share_offload_lifecycle 在 base 上失败，在本 PR 上通过，证明问题真实存在且修复有效。

实现拆解

1. 键类型与簿记初始化变更：python/sglang/srt/disaggregation/decode_kvcache_offload_manager.py 的 __init__ 中，self.offloaded_state 与 self.offload_inflight 从无类型 {} 改为显式的 dict[Req, OffloadedState] 与 dict[Req, int]，并补充注释说明原因：调用方可能在响应结束后立刻复用 rid，而旧请求的异步 D2H 拷贝仍在飞。
2. 辅助函数签名切换：_mark_offload_started、_mark_offload_finished、_has_inflight_offload 三个方法的入参从 rid 改为 req: Req，内部字典操作同步改为以 req 为键。这样计数器的增减始终绑定到发起 offload 的原始请求对象。
3. 调用点全量适配：offload_kv_cache 中状态查找与写入（self.offloaded_state.get(req)）、_check_offload_progress 中 ack 处理（_mark_offload_finished(req)、last_hash 更新、_has_inflight_offload(req)）、_release_finished_req 与 finalize_release_on_finish 中的状态读取与删除，全部从 req.rid 切换为 req；_release_finished_req 末尾的 if req.rid in ...: del ... 简化为 self.offloaded_state.pop(req, None)。由于 ongoing_offload[ack_id] 中本就保存着确切的 req 对象，ack 回调天然能拿到正确的实例，无需任何接口变更。
4. 测试配套：test/registered/unit/disaggregation/test_specv2_kvcache_offloading.py 中所有对 offloaded_state 与 offload_inflight 的访问从 req.rid 改为 req，并新增回归测试

test_reused_rid_does_not_share_offload_lifecycle，构造两个 rid 相同但实例不同的请求，验证旧请求的迟到 ack 只能清理自身状态、新请求的 offload 进度与计数不受影响。

关键文件：

- python/sglang/srt/disaggregation/decode_kvcache_offload_manager.py（模块 卸载管理；类别 source；类型 core-logic；符号 _mark_offload_started, _mark_offload_finished, _has_inflight_offload）：核心修复文件：将 offload 生命周期状态（offloaded_state、offload_inflight）的键从 rid 改为 Req 实例，覆盖初始化、三个辅助函数、offload_kv_cache、_check_offload_progress、_release_finished_req 和 finalize_release_on_finish 的全部访问点。
- test/registered/unit/disaggregation/test_specv2_kvcache_offloading.py（模块 单元测试；类别 test；类型 test-coverage；符号 test_reused_rid_does_not_share_offload_lifecycle）：测试配套：将既有测试中的状态访问全部改为按 Req 实例键，并新增 test_reused_rid_does_not_share_offload_lifecycle 回归测试，覆盖同 rid 双请求场景下迟到 ack 的状态隔离。

关键符号：_mark_offload_started, _mark_offload_finished, _has_inflight_offload, _release_finished_req, finalize_release_on_finish, test_reused_rid_does_not_share_offload_lifecycle

关键源码片段

python/sglang/srt/disaggregation/decode_kvcache_offload_manager.py

核心修复文件：将 offload 生命周期状态（offloaded_state、offload_inflight）的键从 rid 改为 Req 实例，覆盖初始化、三个辅助函数、offload_kv_cache、_check_offload_progress、_release_finished_req 和 finalize_release_on_finish 的全部访问点。

```
# 核心修复摘要：offload 生命周期状态全部改按 Req 实例做键

class DecodeKVCacheOffloadManager:
    def __init__(self, ...):
        ...
        # 之前这里是无类型的 {}（按 rid 字符串键），现在显式以 Req 实例为键。
        # 调用方可能在响应结束后立刻复用 rid，而旧请求的异步 D2H 拷贝仍在飞，
        # 键改为 Req 实例后，同 rid 的不同请求天然隔离。
        self.offloaded_state: dict[Req, OffloadedState] = {}
        self.offload_inflight: dict[Req, int] = {}

    # 入参从 rid 改为 req。offload_kv_cache 与 ack 回调都持有确切的 Req 实例，
    # 因此不会出现同 rid 跨请求的错误命中。
    def _mark_offload_started(self, req: Req):
        self.offload_inflight[req] = self.offload_inflight.get(req, 0) + 1

    def _mark_offload_finished(self, req: Req):
        count = self.offload_inflight.get(req, 0)
        if count <= 1:
            self.offload_inflight.pop(req, None)
```

```
            else:
                self.offload_inflight[req] = count - 1

        def _has_inflight_offload(self, req: Req) -> bool:
            return self.offload_inflight.get(req, 0) > 0

        def _check_offload_progress(self, finish_count):
            """处理设备到主机的 offload 完成事件（ack）。"""
            cc = self.cache_controller
            while finish_count > 0:
                ack = cc.ack_write_queue.pop(0)
                ack.finish_event.synchronize()
                for ack_id in ack.node_ids:
                    (req, host_indices, incremental_tokens, start_time, start, end) = \
                        self.ongoing_offload.pop(ack_id)

                    # 这里拿到的是发起 offload 时的原始 Req 对象，而不是通过 rid 反查，
                    # 因此迟到的 ack 只会作用于真正属于它的请求，不会污染同 rid 的新请求。
                    self._mark_offload_finished(req)
                    prior_hash = (
                        self.offloaded_state[req].last_hash
                        if req in self.offloaded_state else None
                    )
                    last_hash = self._trigger_backup(
                        req, host_indices, incremental_tokens, start_time, prior_hash
                    )
                    if req in self.offloaded_state:
                        self.offloaded_state[req].last_hash = last_hash

                    if req.finished() and not self._has_inflight_offload(req):
                        state = self.offloaded_state.get(req)
                        start_offset = state.prefill_len if state is not None else start
                        self._release_finished_req(req, start_offset)
                finish_count -= 1
```

test/registered/unit/disaggregation/test_specv2_kvcache_offloading.py

测试配套：将既有测试中的状态访问全部改为按 Req 实例键，并新增 test_reused_rid_does_not_share_offload_lifecycle 回归测试，覆盖同 rid 双请求场景下迟到 ack 的状态隔离。

```
    # 回归测试：两个请求共享同一个 rid，但 Req 实例不同。
    # 第一个请求的 D2H 拷贝尚未完成时，迟到的 ack 不能修改第二个请求的状态。
    def test_reused_rid_does_not_share_offload_lifecycle(self):
        manager, _ = _make_manager(pool_size=32, page_size=4)
        manager.cache_controller = MagicMock()
        manager.cache_controller.get_hash_str.return_value = 'prefill_hash'
        manager.cache_controller.write.return_value = torch.arange(4, 8, dtype=torch.int64)
        manager.decode_host_mem_pool = MagicMock()
        manager.request_counter = 0
```

```
manager.offload_stride = 4

old_req = _make_mock_req(
    req_pool_idx=0, kv_committed_len=20, kv_allocated_len=20, rid='reused'
)
new_req = _make_mock_req(
    req_pool_idx=0, kv_committed_len=20, kv_allocated_len=20, rid='reused'
)
for req in (old_req, new_req):
    req.origin_input_ids = [0, 1, 2, 3]
    req.output_ids = [4, 5, 6, 7, 8]
    req.finished.return_value = False

self.assertTrue(manager.offload_kv_cache(old_req))
old_req.finished.return_value = True

# 请求结束并不等于异步拷贝完成，调用方可能立刻重用 rid。
self.assertTrue(manager.offload_kv_cache(new_req))
self.assertIsNot(old_req, new_req)
self.assertIn(old_req, manager.offloaded_state)
self.assertIn(new_req, manager.offloaded_state)
self.assertEqual(manager.offload_inflight[old_req], 1)
self.assertEqual(manager.offload_inflight[new_req], 1)

# 仅让旧请求的 ack 到达，模拟迟到回调。
manager.cache_controller.ack_write_queue = [
    HiCacheAck(None, _FinishedEvent(), [1])
]
manager._trigger_backup = MagicMock(return_value='old_last_hash')
manager._check_offload_progress(1)

# 旧请求被清理，新请求的状态与计数保持完好。
self.assertNotIn(old_req, manager.offloaded_state)
self.assertNotIn(old_req, manager.offload_inflight)
self.assertIn(new_req, manager.offloaded_state)
self.assertEqual(manager.offloaded_state[new_req].inc_len, 4)
self.assertEqual(manager.offload_inflight[new_req], 1)
self.assertIn(2, manager.ongoing_offload)
```

评论区精华

本 PR 没有实质性的 review 评论（review_comments_count=0），仅有合并干系人的 CI 操作记录：

> hnyls2002 发起 /rerun-test registered/unit/disaggregation/test_specv2_kvcache_offloading.py registered/disaggregation/test_disaggregation_decode_offload.py
> github-actions[bot] 回报：ubuntu-latest 上的单元测试与 2-gpu-h100 上的 E2E decode offload 测试均通过。

hnyls2002 最终给出 APPROVED。PR body 中附带的确定性复现说明（只恢复回归测试到 base 提交即可稳定复现失败）是本 PR 最有说服力的论证，弥补了 review 讨论的缺失。

- 确定性的回归复现方式 (test): 修复后同一测试通过，复现路径与修复路径一一对应，验证了改动的有效性。
- CI 重跑范围与批准 (other): 两组测试均通过，修复在 mock 单测与真实两卡 E2E 场景下均得到验证。

风险与影响

- 风险：
  1. Req 哈希 / 相等语义依赖：改用 Req 实例做字典键，假设 Req 保持默认的对象身份哈希与相等语义。若未来 Req 实现基于 rid 的自定义 __eq__/__hash__，本修复将重新退化为按 rid 隔离，需要额外测试守护。当前测试中两个同 rid 请求实例被视作不同键，间接验证了默认语义。
  2. 对象引用生命周期：offloaded_state 与 offload_inflight 持有 Req 引用直到请求释放或 ack 处理完毕，相比原字符串键引用更重，但释放路径（_release_finished_req 的 pop 与 ack 后的计数清零）已覆盖，正常请求不会长期滞留。
  3. 竞态修复深度：本 PR 修复了“键碰撞”层面的串扰，但 _check_offload_progress 中仍依赖 ack.finish_event.synchronize() 后才处理 ack；若并发调度下出现乱序 ack，状态机仍依赖基线的队列顺序保证。代码中未发现引入新的并发风险。
  4. 测试覆盖边界：新增回归测试是纯 mock 单测，未覆盖真实 CUDA 事件与多 TP rank 场景；E2E 测试 test_disaggregation_decode_offload.py 已由合并者重跑通过，但长期建议补充压力场景下 rid 高频重用的集成测试。 - 影响：影响范围集中在解码侧 KV offload（HiCache）路径：DecodeKVCacheOffloadManager 的所有调用方（disaggregation decode 流程）无需改动接口即可获得正确行为。修复消除了因 rid 重用导致的陈旧进度继承与迟到回调污染，避免 KV 内容错乱、host 池提前释放或双重释放等资源隐患。对不使用 HiCache 或 offload 功能的用户无影响。团队测试资产同步更新，所有相关单测改为按 Req 访问状态，后续开发者必须遵循这一约定。 - 风险标记：核心路径变更，依赖 Req 哈希语义，异步竞态修复，回归测试覆盖竞态核心

关联脉络

- PR #37166 fix(staging): make empty staging rings reusable: 同为 disaggregation 路径的异步状态稳定性修复，修复空暂存环导致的 PD 暂存转发卡死，与本 PR 同属一次对异步资源生命周期问题的集中治理。
- PR #37194 [Fix] Shut hicache test servers down gracefully before SIGKILL: 同样围绕 HiCache 测试基础设施的稳定性修复，与本 PR 的 hicache decode offload 测试同处一条 CI 测试链路。
- PR #35281 [PD] Align defensive protocol behavior across Mooncake, NIXL, and Mori: 对齐三个 PD 后端的防御性协议行为并修复竞态与 KV 布局缺陷，与本 PR 同属 disaggregation 正确性加固方向。

- PR #37094 [mem_cache] Move req_pool_idx into ReqKvInfo: 将请求池索引等状态归入 ReqKvInfo，体现“请求生命周期状态归属到具体对象”的设计趋势，与本 PR 将 offload 状态绑到 Req 实例的思路一致。